Development of a state machine sequencer for the Keck Interferometer: evolution, development & lessons learned using a CASE tool approach

Leonard J. Reder[*a], Andrew Booth[*a], Jonathan Hsieh[*a], Kellee Summers[†b]

[a]Jet Propulsion Laboratory, California Institute of Technology, [b]W. M. Keck Observatory, California Association for Research in Astronomy

ABSTRACT

This paper presents a discussion of the evolution of a sequencer from a simple Experimental Physics and Industrial Control System (EPICS) based sequencer into a complex implementation designed utilizing UML (Unified Modeling Language) methodologies and a Computer Aided Software Engineering (CASE) tool approach. The main purpose of the Interferometer Sequencer (called the IF Sequencer) is to provide overall control of the Keck Interferometer to enable science operations to be carried out by a single operator (and/or observer). The interferometer links the two 10m telescopes of the W. M. Keck Observatory at Mauna Kea, Hawaii.

The IF Sequencer is a high-level, multi-threaded, Harel finite state machine software program designed to orchestrate several lower-level hardware and software hard real-time subsystems that must perform their work in a specific and sequential order. The sequencing need not be done in hard real-time. Each state machine thread commands either a high-speed real-time multiple mode embedded controller via CORBA, or slower controllers via EPICS Channel Access interfaces. The overall operation of the system is simplified by the automation.

The UML is discussed and our use of it to implement the sequencer is presented. The decision to use the Rhapsody product as our CASE tool is explained and reflected upon. Most importantly, a section on lessons learned is presented and the difficulty of integrating CASE tool automatically generated C++ code into a large control system consisting of multiple infrastructures is presented.

Keywords: Interferometer, sequencer, CASE tool, UML

1. INTRODUCTION

The Keck Interferometer is the major ground based instrument of NASA's Origins program. The goal of the program is to search for extra-solar planets, measurement of extra solar zodiacal dust, and general high angular resolution IR astrophysics and imaging. Since June of 2002 science data has been collected utilizing the Interferometer (IF) Sequencer for high-level instrument control. The IF Sequencer is implemented using a subset of the UML design methodology (two out of twelve UML views). The commercial Rhapsody CASE tool product provided by I-logix is used for graphical entry of the design and to automatically generate C++ code. Thus the C++ generated is coupled to Rhapsody's UML model. To understand the IF Sequencer structure one must first have a conceptual knowledge of the sequencer role within the overall Interferometer control system software architecture.

The entire control system software consists of a hierarchy of state based controllers (figure 1)[1]. These consist of various high-speed real-time embedded controllers based on a JPL framework developed specifically for real-time Interferometer control (known as the "JPL RTC Toolkit"[2] or within this paper referred to as "RTC"). Slower

[*] Leonard.J.Reder@jpl.nasa.gov; phone 1 818 354 3639; fax 1 818 393 4357; Jet Propulsion Laboratory, MS171-113, 4800 Oak Grove Drive, Pasadena, CA 91109

[†] ksummers@keck.hawaii.edu; phone 1 808 885 7887; fax 1 808 885 4464; W. M. Keck Observatory, 65-1120 Mamalahoa Highway, Kamuela, HI 96743

functionality within the system utilizes legacy Keck Observatory telescope control infrastructure built on the Experimental Physics and Industrial Control System[‡] (EPICS). The overall operation of the system is provided by automation implemented in the IF sequencer at the top-level. The IF Sequencer is a software program designed to command the various objects within RTC (that provide real-time servo control of subsystems components such at Fast Delay Lines (FDLs), Fringe Trackers, etc.). High-level commands are sent to the two large Keck telescopes by way of telescope sequencers that isolate the IF sequencer from the complexity of the various Keck telescope subsystems. All components must be commanded to perform their work in a specific and sequential order; though not in the hard real-time domain. The IF Sequencer is the highest level of control in the Keck Interferometer software control system.[§]

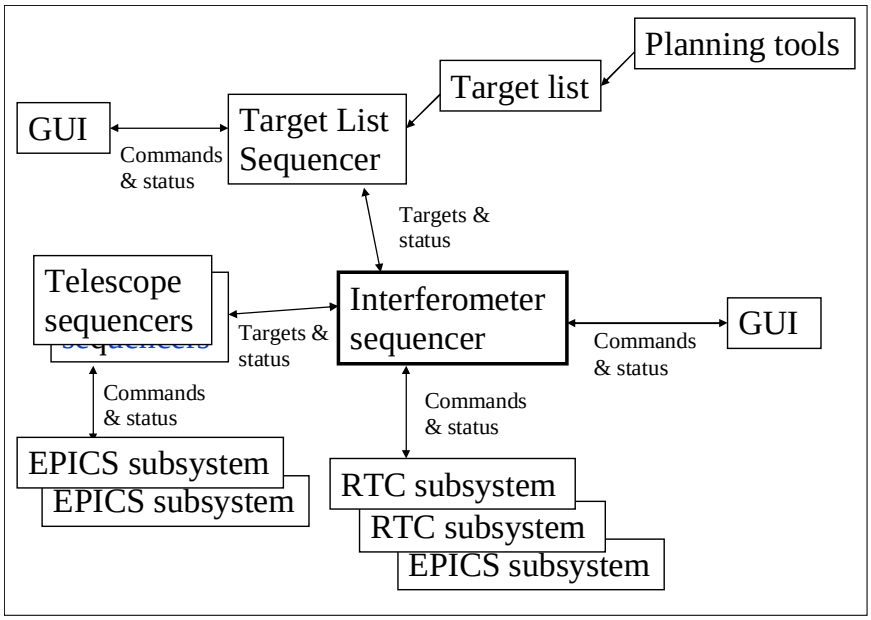

Fig. 1 Sequencing hierarchy

In the following sections of this paper the development from a simple EPICS implementation to the object oriented design currently being deployed at Keck are discussed. Our evaluation and selection of the Rhapsody CASE tool (which was essentially motivated by the automatic state machine code generation capability) is presented. Finally, some lessons learned and conclusions are given.

2. HISTORY

During the development of RTC and subsequent implementation of Keck Interferometer specific software there had been much discussion of how to control the lower level subsystems in a unified and consistent way. Experience with Palomar Testbed Interferometer (PTI) suggested that a high level control program be developed. At PTI, a sequencer was implemented to run within the VxWorks operating system to perform overall control of the system.

The W. M. Keck Observatory had been using State Notation Language (SNL) that runs within a general sequencer control program as part of EPICS. This sequencer program was being used for Keck motion control as part of the Keck AO system and elsewhere. At the start of the IF Sequencer development, the EPICS sequencer had been recently ported

[‡] More information on the Experimental Physics and Industrial Control System infrastructure is at http://mesa53.lanl.gov/lansce8/EPICS/ or http://www.aps.anl.gov/epics/.

[§] The current implementation of the Interferometer sequencer includes a primitive version of the Target List Sequencer functionality implemented between the IF sequencer C++ and the GUI. The Target List Sequencer is intended to further automate the systems and allow timed observations of groups of star targets. At the time of this writing the implementation of the Target List Sequencer has not begun.

to UNIX and was a viable solution to the sequencing problem. There was also a desire to use the W. M. Keck legacy approach and existing organizational standards. Due to a requirement to quickly demonstrate first fringes operation of the instrument an initial version had to be rapidly implemented. The first version (Increment 0) of the IF Sequencer was a simple UNIX EPICS sequencer written in SNL that generated periodic sidereal delay values corresponding to a star target being observed. These values were sent to RTC developed Fast Delay Line controllers via CORBA commands. Since the SNL language is only an extension to conventional C, the implementation required wrapping CORBA interfaces with a simple C code API. A library for computing delay line target values at PTI was reused. And a simple TCL GUI was written that used legacy Keck Keywords to command and monitor the Increment 0 EPICS based IF Sequencer. This scheme was successfully used for acquiring first fringes but did not have the functionality required for full up science operation.

It was quickly realized that building state machines purely from procedural compiled code would be cumbersome and not scale well with complexity, so we set out to explore other solutions. First we tried a more object object-oriented approach by using TCL with the object-oriented extension called Incr TCL. A small test sequencer consisting of a simple finite state machine was coded in TCL to drive two test siderostats that were in place at Keck for testing. We used the State design pattern of reference [3] as the basis for the script. Although using a scripting language made it easy to modify and test, again, it was realized that with more complex state machines, the code would become difficult to maintain. It was also noted that some additional development effort would be required to implement the multi-threaded framework that would be required.

At about the time we were considering these solutions, a colleague suggested using a CASE tool to automatically generate state machine codes by first representing them as UML state charts. The JPL Deep Space One technology demonstration mission had evaluated the use of a product called Rhapsody (manufactured by I-logix Corp.[4]). We examined the I-logix documentation and quickly decided that Rhapsody was worth serious consideration. After seeing an on-site presentation and demonstration we started to build basic evaluation sequencer models. This proved to be more difficult than expected, but without other options, we decided using Rhapsody as our development tool was an improvement over hand coding. We will comment more about the evaluation in the next section, but from the initial prototype work, we were able to build the first deployed IF Sequencer (Increment 1) for the purposes of first science operations.

The Increment 1 version consisted of UML models entered into Rhapsody; only the static class diagram and state chart views were used. C++ code was auto-generated for execution on the Solaris operating system since real-time was not a requirement. The Rhapsody tool was coupled to the ACE/TAO CORBA ORB that RTC is built on and a set of helper classes to wrap primitive RTC functionality were designed. Included in these helper classes was the implicit decision that the IF Sequencer would be hierarchical with some sort of communications infrastructure sending messages from one state machine to another. The idea adopted early on was that for each Interferometer subsystem there would be first a mid-level state machine for direct control (thus we have a state machine for each major Interferometer low-level subsystem: Fringe Tracker, Fast Delay Line, Angle Tracker, etc.) and then a high-level state machine which commands each of the mid-level ones.

During the initial development using Rhapsody, a parallel effort was going on to port RTC to the Linux-RTAI platform in summer of 2001[5]. The RTC had been evolving as well. Eventually there was a port to the Sun Solaris operating system. It was therefore logical that our next version (Increment 2) leverage and reuse the newest RTC infrastructure (Fig. 2). RTC consists of a collection of libraries and three executable programs. The libraries provide a white box framework for building specific servo control objects. The framework includes support for configuration and telemetry. The executable programs are the Telemetry and Configuration Servers and a CPU Manager. The Telemetry Server provides a publish/subscribe telemetry implementation via CORBA event channels. A Configuration Server provides a link to a database of persistent storage so that any parameter within the IF Sequencer can be independently configured at run-time (Increment 1 used a flat file configuration independent of the RTC Configuration Database).

Figure 2 shows a UML component view of the IF Sequencer within RTC. The use of the CPU Manager allows one to load and instantiate specific objects that contain state machine code. The CPU Manager is CORBA enabled as are all IF Sequencer state machine objects (known as RTC Managed Objects since they are loaded and instantiated by the CPU

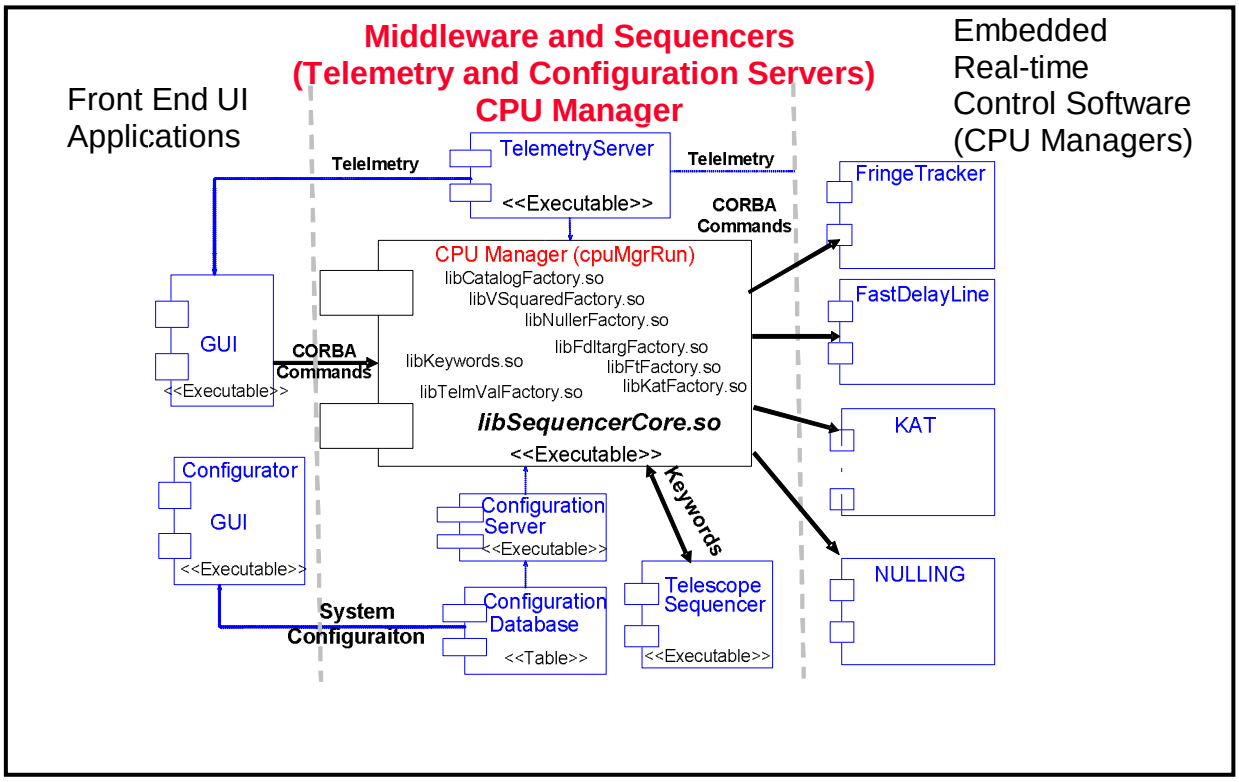

Fig. 2 Component diagram of IF Sequencer Configuration with JPL RTC Toolkit

Manager). The advantage of this scheme over the Increment 1 is that object instantiation is no longer hardwired. The application can be reconfigured for various modes of operation. This makes reconfiguration for Visibility Squared, Astrometry, Nulling, Differential Phase and Imaging modes of the instrument possible. More detail on the specific UML object-oriented framework is presented in section 4.

3. EVALUATION

Initially, our major goal was to automatically generate state machine code, and we had a rather naïve view of what CASE tool technology was and what it could do. So immediately a simple, proof of concept, Fringe Tracker state machine sequencer was implemented. A great deal was learned about the tool and the prototype Fringe Tracker state machine became the basis for our UML model framework.

Immediately it was discovered that Rhapsody was much more then than a simple state machine code generator. There was an initial conception that non-programmers would easily use the tool but this was quickly dismissed to be impractical. Moreover, Rhapsody is, as we quickly learned, a UML design and code generation tool in a general sense. Rhapsody supports the entire set of UML diagram types[8]. Figure 3 shows a screen snapshot of Rhapsody. The UML state chart (a high-level state-machine visibility squared sequencer) is in the right hand window and an overall tree view of the code model is in the left window. It was learned that groups of classes could be further designed graphically from which thousands of lines of code were automatically generated.

Code generation from a conceptual graphical model is inherently a tricky thing to do and, it gets even trickier when third party legacy frameworks must be integrated. Issues arise that you would never think of when hand-writing code. There are specific manglings that one might want to inflict upon source in special circumstances. Rhapsody-generated source has its own “coding style”. Unfortunately, this style is inherently different from that of RTC. Modification of the code style can be made based on adjustment of a large number (hundreds) of configurable properties within the

Rhapsody product. The properties allow one to change everything from the graphical appearance of the model to how code is automatically generated and more. Although the scheme provides great flexibility when making modifications to generated code, it also causes an equal amount of confusion to the developer trying to change something specific about the source code. To modify something in the auto-generated code requires the correct property to be changed. It is a nice feature in that properties follow the hierarchy of your model in scope, but they still can be difficult to keep track of. For all the inconvenience, the Rhapsody tool is very flexible, and without this feature we would not have been successful in integrating Rhapsody with our existing RTC framework classes.

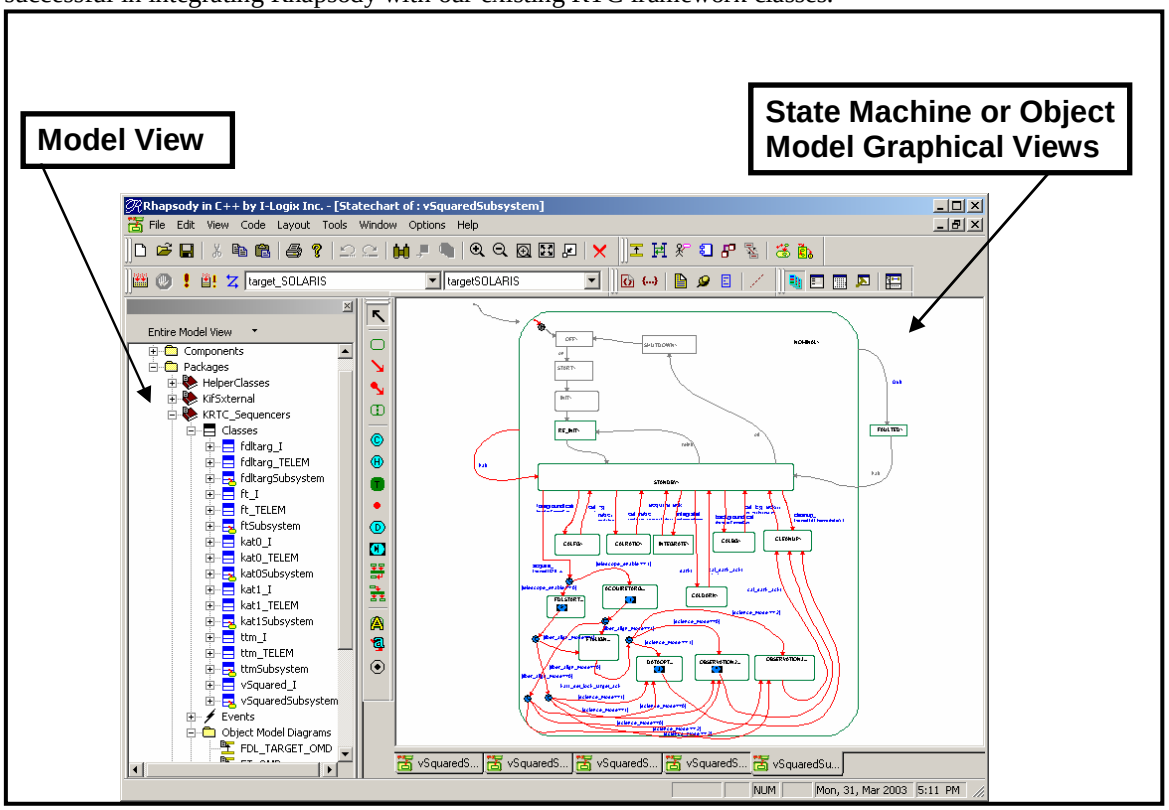

Fig. 3 Screen shot of Rhapsody CASE Tool showing model view on left and Harel statechart of visibility squared science observation sequence state machine subsystem class on right.

The process of using the Rhapsody tool for us was to first enter UML and then spend much time exploring the properties and entering code within the UML that implements interfaces and uses the Rhapsody supplied framework. Our core functionality of state machines is embedded into particular classes, the details of which are discussed in the next section. Within these state machines, one would select states and enter code into specific text entry areas; the resulting generated code would contain this hand-entered code in specific areas.

Creating code using the methodology of first entering graphical representations to generate framework code and then entering particular fragments of code to provide specific interfaces and algorithm functionality is quite daunting and extremely difficult for the developer that is accustomed to line-by-line coding. With much patience and work one can become proficient with the technology. So our evaluation resulted in the conclusion that auto-generating code for implementing state machines was not by any means easy, but would be more appropriately characterized as doable and more manageable over the entire life cycle of our software than any of the other approaches we tried.

4. OBJECT ORIENTED DESIGN (THE UML MODEL)

Figure 2 shows a conceptual component view of the IF Sequencer as it exists within RTC. At the far right are the Keck specific components that provide fast servo control directly to Interferometer hardware. These are running on VxWorks

single board computers as RTC CPU Manager tasks. Each of these VxWorks components is a set of objects known in RTC as "Gizmos".

The IF Sequencer basic functions are (1.) sending commands to RTC Gizmos (e.g. lower-level subsystems) via CORBA and (2.) transmitting and receiving data through the RTC Telemetry Server (top center of Fig. 2). The sequencer monitors telemetry items in order to detect responses from subsystems that were commanded. Telemetry is generated by the IF Sequencer to update status on front-end GUIs that consist of a Python script (left side of Fig. 2).

The Configurator GUI/Configuration Server/Configuration Database as shown in Fig. 2 are used to provide dynamic run-time configurability to the Sequencer. At the center of Figure 2 is the RTC CPU Manager component. Within the CPU manager framework are standard interfaces for dynamically loading shared libraries, then finding and/or creating instances of objects defined by these libraries. The implementation of the IF Sequencer consists of a set of shared libraries. All of the state machines and support infrastructure are compiled in to a single library called libSequencerCore.so and into several different lib*Factory.so libraries. Normally, the objects would have been separated into individual libraries but because we are using the Rhapsody supplied event communication framework rather than CORBA for state-machine object to state-machine object communications, all objects had to be consolidated into a single shared library. For each RTC Managed object, we implemented a factory library that provided the interfaces needed by the CPU Manager framework.

The Sequencer is organized as a hierarchy of classes both in a static sense and in a dynamic sense. Figure 4 shows the static class diagram. Each of the most specialized classes is a standalone state machine. The state machine code is defined by user entered statecharts that are associated with a particular class; C++ state-machine code is then automatically generated. State-machine objects have names ending in suffix of "Subsystem".

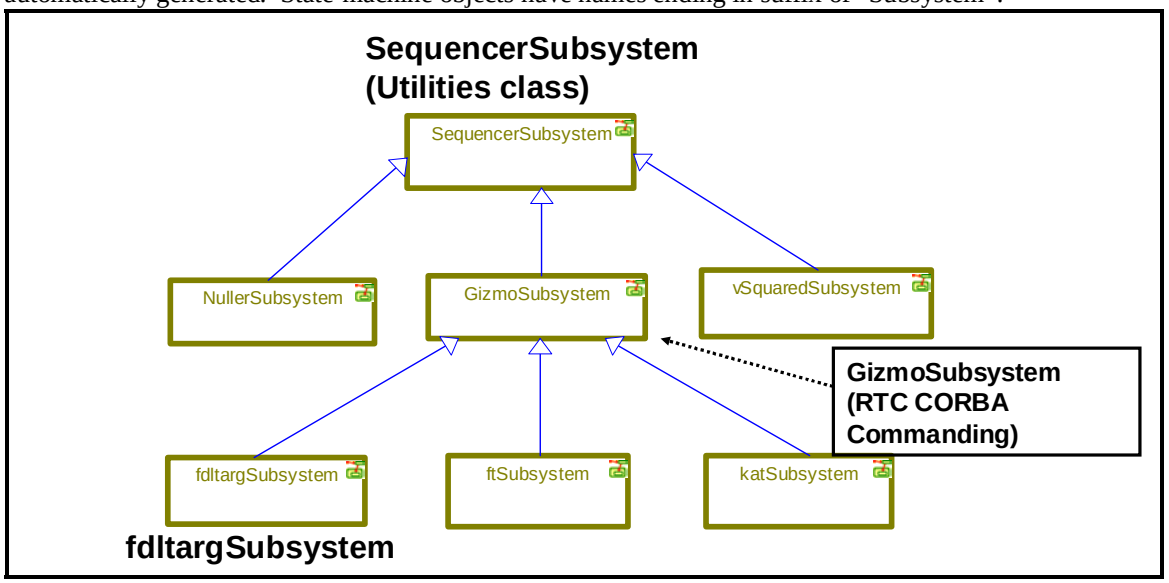

Fig. 4 Static class diagram of subsystems class hierarchy

The Sequencer is composed of several mid-level subsystem classes controlled by a single high-level state machine class. Each of the mid-level subsystem classes, such as **fdltargSubsystem** (the FDL sidereal target generator), is itself a state machine that acts as the interface to one or more RTC Gizmos. There is one high-level state machine for each science mode of the Keck Interferometer. Currently, only the Visibility-Squared mode (**vSquaredSubsystem**) state machine has been implemented. A new high-level state machine for Nulling operations (**NullerSubsystem**) is in its initial stages of development at the time of this writing. Each of the mid-level state machines has been designed with reusability and extensibility in mind. The same **fdltargSubsystem** class used by the **vSquaredSubsystem** can be instantiated for use with a high-level **NullingSubsystem** class with no modifications. As more observing modes come online, more high-level state machines will be implemented. Similarly, as more RTC Gizmos are implemented, more mid-level subsystem objects will be needed.

There are many commonalities among the Subsystem classes that make up high-level and mid-level subsystem classes. In Fig. 4 the **SequencerSubsystem** and **GizmoSubsystem** classes are utility classes that implemented the supporting functionality that enables subsystem classes to send events, push and monitor various types of telemetry and find, connect to and command an RTC Gizmo.

Although our current implementation only has two levels of subsystem classes, there can be more levels created if desired. A hierarchical approach seemed to make sense since the sequencer functionality always is expressed as some composition of subsystems. Further, a hierarchical (tree) model promotes less confusion in an event model (such as the one included with Rhapsody) because events can only be passed up and down the tree, not sideways.

4.1 SequencerSubsystem Class

In the IF Sequencer, the **SequencerSubsystem** class provides the foundation for all subsystem (state-machine) objects. It is a concrete base class, and all other subsystem classes are derived from this class. Figure 5 shows the **SequencerSubsystem** class dependencies. Note that the **Collection**, **ManagedObject** and **ManagedObjectImpl** classes are not actually implemented within the Rhapsody UML model view of Fig. 5 but are stubs referencing externally-implemented code within the core RTC shared library. The two classes stereotyped as <<CORBAInterface>> are IDL[6] (Interface Definition Language) that define top-level interface methods that all subsystem classes contain.

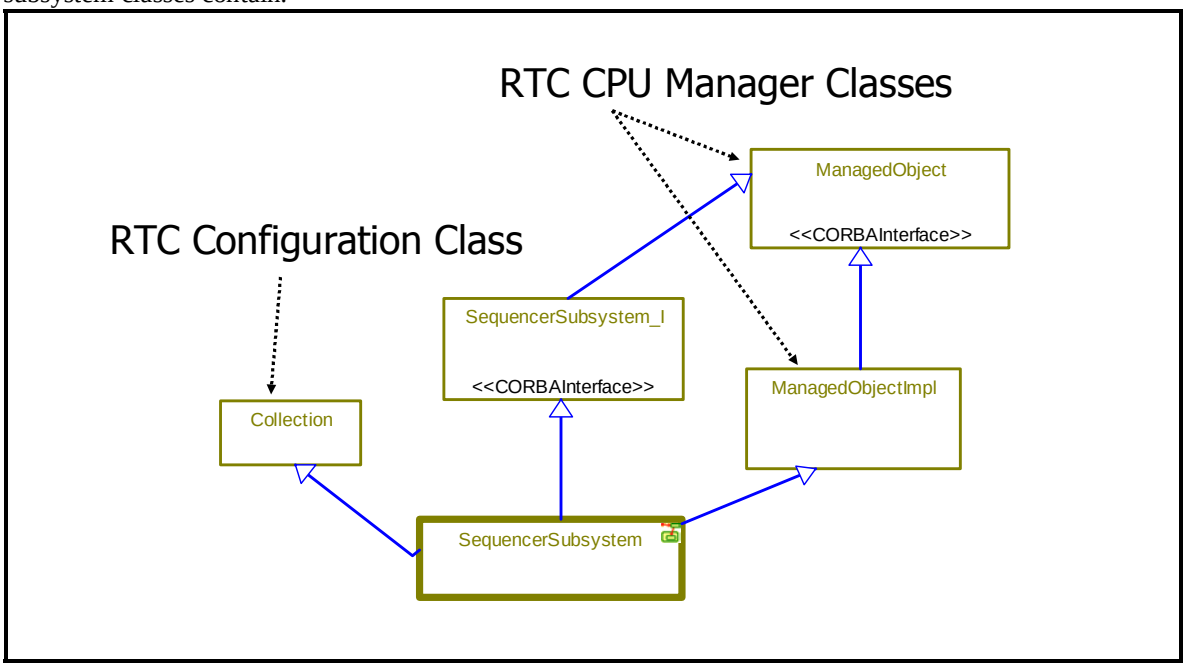

Fig. 5 SequencerSubsystem utility class relationship to RTC Toolkit.

The **SequencerSubsystem** class defines a state machine that serves as the basis for building application-specific subsystem classes (shown on Fig. 6a). First we describe the utility functions implemented within the **SequencerSubsystem** class.

SequencerSubsystem defines several methods to implement a subsystem hierarchy of event communication. The rule we adopted early on was that the topology is restricted to a single parent and any number of child **SequencerSubsystems**. Rhapsody events are passed to the parent through the ***Upstream*** method and broadcast to all children through the ***Downstream*** method. The purpose of the ***Upstream*** method is to inform a higher-level parent state machine of important events such as exceptions without knowing the exact type of the parent. Thus, an **fdltargSubsystem** instance can propagate a fault event up the hierarchy without knowing if its parent is a

vSquaredSubsystem or a **NullerSubsystem**. The ***Downstream*** method enables the opposite functionality: broadcasting an event to all child subsystems regardless of exact type. For example, a **vSquaredSubsystem** can send an "on" event to all of its connected children. Finally, ***PushStatusTelemetry*** is used by GUIs or other clients to update data on the internal state of the **SequencerSubsystem**. This method pushes the most recent value for all attached telemetry items. It is also called after an object has been configured. This way, client applications are informed of configuration changes as soon as they happen.

The **SequencerSubsystem** class inherits from **Collection** to support RTC Configuration use and from **ManagedObjectImpl** in order to be interfaced with the RTC CPU Manager framework.

4.1.1 RTC Managed Objects

Every subsystem class (Fig. 5) within the IF Sequencer is derived from **ManagedObjectImpl** and has an associated factory class derived from **ObjectFactoryImpl** that makes them RTC Managed objects. Within the framework, objects are created by factories. When the managed object is created by the factory, a unique Id is registered with an **ObjectManager** (another RTC object) within the CPU Manager. That Id contains both instance Name and Type information, which are used everywhere the subsystem needs a Name or Type reference. Thus an object already created within the CPU Manager can readily be found.

4.1.2 Configurable Objects (Collection Class)

Figure 5 shows that the **SequencerSubsystem** class inherits from RTC **Collection** class. The **Collection** class provides an aggregation relationship with two RTC template convenience classes; **Entry<*>** and **Array<*>.** These classes provide the interface to the Configuration Server. The template type argument can be any of the IDL defined primitive types. When a new configurable parameter is desired, a new attribute of type **Entry<*>** or **Array<*>** is added to the subsystem class model. The attributes act like conventional primitive types but are configurable.

The Configurator GUI at the bottom left of Fig. 2 is used to set values and execute reconfiguration of subsystem object instances within the IF Sequencer.

4.1.3 CORBA Command Bindings

Each of the mid-level subsystem classes inherit from the **GizmoSubsystem** class that provides the capability to resolve and bind references to RTC Gizmos registered in a CORBA Name Service. The **GizmoSubsystem** class has an association with a global singleton running asynchronously in a separate thread called the **GizmoManger**. The **GizmoManager** contains a pair of methods for connection and reconnection to RTC Gizmos. Each class derived from **GizmoSubsystem** overrides the pure virtual ***instrumentInit*** method, which calls a ***`bindGizmo`*** method of the **GizmoManager**. This method will asynchronously find a Gizmo in the CORBA Name Service and correctly bind the reference to the **GizmoSubsystem** data member. If at any time the Gizmo becomes unavailable, **GizmoSubsystems** calls the **GizmoManager** ***`reBindGizmo`*** method, which continuously attempts to find the Gizmo again. Since **GizmoManger** is running in a separate thread, the subsystem object thread will never be blocked with connection retries.

4.1.4 Telemetry

The IF Sequencer utilizes the RTC publish/subscribe telemetry infrastructure via an intuitive, efficient, multi-threaded interface to the Telemetry Server for both supplying and consuming telemetry items. **SequencerSubsystem** derived classes publish telemetry in order to update status on GUIs and in archiver software. Subsystem classes subscribe to telemetry items to monitor behavior of RTC Gizmos under control. Telemetry channel names are hierarchical.

4.1.4.1 Supplying Telemetry

New instances of an RTC object of type **SupplierImpl** and/or Struct**SupplierImpl** are the fundamental client side objects used to connect to a unique telemetry channel (one of these objects is instantiated for each channel named). The overloaded assignment operator "=" is then used to publish (or push) telemetry based on a publishing mode. For example, one can instantiate a **SupplierImpl** reference called MyTelemetrySupplier and then push MyValue using the statement "MyTelemetrySupplier = MyValue;". The publishing mode is set at construction time and can be changed externally by configuration. Currently the publish modes supported include: OFF, (the publishing is disabled), FULL_RATE (published at full rate), VALUE_CHANGE (sending of telemetry over channel happens only on a value change) or SAMPLE (publish a single value at fixed interval of time).

4.1.4.2 Consuming Telemetry

In RTC, the final destination of telemetry is a user-defined filter class derived from a RTC **Filter** (handler) class. Consuming modes can be set analogous to the publishing (supplier) mode of **SupplierImpl** instances. The telemetry infrastructure uses a push/push (CORBA event) channel model that allows subscribers to passively wait for data (i.e. monitor). The subscriber to a telemetry channel defines a ***Push*** method within one's handler. The methods will be called whenever telemetry moves through the subscribed channel. The ***Push*** methods, in the IF Sequencer, are typically implemented to generate Rhapsody events that cause a state machine (subsystem object) to respond to the received telemetry value. An example of this is the Fringe Tracker state machine class (**ftSubsystem**) where an aggregation relationship to a **LockSecsFilter** class is established. The **LockSecsFilter** class inherits from Filter class and implements a ***Push*** method that tests telemetry representing the amount of time a Fringe Tracker has been locked onto a fringe during an observation. When this exceeds an internally set time, an event is generated to the **ftSubsystem** state machine that commands it to sequence the Fringe Tracker to stop tracking.

It is important for consumers to process incoming telemetry data quickly so as not to tie up the limited resources of the Telemetry Server. This is accomplished through the use of an AsynchronousDispatcher to handle telemetry distribution with a separate pool of threads, which effectively decouples the Telemetry Server from any consumers.

4.1.5 Derived Classes

Figure 4, above, shows the set of derived classes in the IF Sequencer. These are the **NullerSubsystem** and, **vSquaredSubsystem** high-level automation classes (observation mode state machines) and the **fdltargSubystem**, **ftSubsystem** and **katSubsystem** mid-level automation classes (for direct sequencing control of RTC Gizmos). More will be developed as needed. The **fdltargSubsystem** delivers pre-computed sidereal targets (delay settings) to update FDL (fast delay line) positions and control them; the **ftSubsystem** controls the FT (fringe tracker) and the **katSubsystem** controls the (Keck) angle trackers. Each of these implements a specialized state machine (subsystem) class (e.g. **fdltargSubsystem** in the example shown in Fig. 6b) that is derived from the base state machine (Fig. 6a) implemented within the **SequencerSubsystem** class. Derived classes automatically contain the basic state chart functionality.

The base state machine (Fig 6a) contains five states within a NOMINAL composite state to provide standard behavior to all derived state machine objects. The only default behavior in each of these five states is to push a structured telemetry item providing information about current state, last state, and status messages about operation, to an external software component such as a GUI. The external WARNING and FAULTED states are used to handle exception behavior (e.g. error states). When a fault condition occurs user code generates an event which causes the state machine to transition ether to WARNING or FAULTED states. Error recovery (from a FAULTED state) is via operator intervention only (a halt event is generated to return to the NOMINAL state). This scheme was at the user's request and inevitably there will be AI planning and fault recovery methods used on the IF Sequencer problem in the future. Faults are also propagated to higher and lower-level state machines as necessary.

Derived classes define custom CORBA IDL interfaces by inheriting from **SequencerSubsystem_I**. An example of this is shown in Fig. 7 for the **fdltargSubsystem** class. Recall that **SequencerSubsystem_I** inherits from **ManagedObject**,

as shown in Fig 5. Thus **SequencerSubsystem_I** provides all the methods in the interface defined by RTC Managed objects in addition to defining the ***On, Off***, ***ReInit***, ***Halt***, ***SimulateOn***, and ***SimulateOff*** methods as controls of its basic (Fig. 6a) state machine behavior. The ***On***, ***Off***, ***ReInit***, ***Halt*** methods generate Rhapsody events that cause corresponding state transitions. The ***SimulateOn*** or ***SimulateOff*** methods turn on and off, respectively, a simulation test mode implemented in every derived state machine object. The idea of the simulation mode is to provide a capability to unit test every state machine in a stand alone configuration, independent of (and without connection to) the real time system.

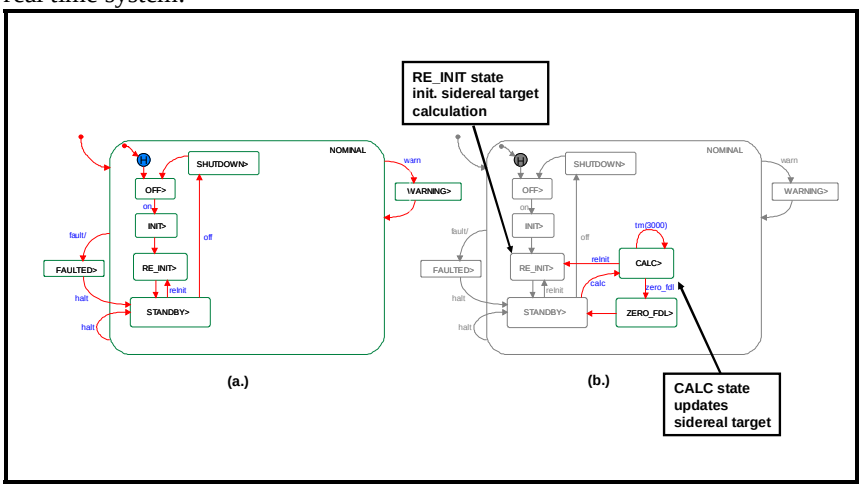

Fig. 6 Harel State charts (a.) base state machine implementation, (b.) FDL (fast delay line) target generator state machine (note b. inherits a.).

Because each state machine object is loaded and instantiated by the RTC CPU Manager, the Rhapsody-generated state machine objects do not have specific knowledge about how they are related. Rhapsody events are passed from state-machine to state-machine by an inverse invocation scheme. Thus each state machine must have a pointer to the state machine it wishes to send an event to. This means that direct associations from state machine to state machine for the purposes of sending events must be explicitly established. Within our UML we create a static class diagram view similar to that of Fig. 4 but this time it defines associations allowed by type. To establish instance linkages another IDL method called ***LinkSubsystems*** is implemented. ***LinkSubsystems*** accepts a sequence of managed object Ids, which specifies a number of other objects that must be referenced. The caller of ***LinkSubsystems*** can also be configurable so that the hooking up of object instances becomes fully configurable as well (typically the caller is a GUI program or startup script).

4.2 Delay Line Target Generator State Machine Object (fdltargSubsystem)

An example of a mid-level state machine class is the **fdltargSubsystem** shown in Fig. 7. An instance of **fdltargSubsystem** is a flexible state machine that can control single or multiple pairs of fast delay lines. The primary purpose of **fdltargSubsystem** is to compute and update FDL delay target positions every three seconds so that fringe tracking can be maintained on a sidereal target (star) moving across the sky. This functionality is implemented in Fig. 6b. There is also a set of IDL methods defined by **fdltarg_I** (not shown in Fig. 6b) that allow fundamental commanding (e.g. Idle, Calc, Track, etc.) of the FDL RTC Gizmo via **fdltargSubsystem**. The input to the **fdltargSubsystem** class is a catalog record of coordinates for the desired observation that is generated by an external observation planning system[7].

While the **fdltargSubsystem** example is perhaps one of the simplest state machine implementations in the IF Sequencer, observations cannot be made without at least one instance of this state machine running. The state machine implementation (Fig. 6b) adds only two states; CALC and ZERO_FDL. The CALC state updates delay target values for specific star position and executes RTC Gizmo CORBA target updates. This is repeated every three seconds as denoted by the tm(3000) in Fig. 6b, which is an internal Rhapsody timer thread event that causes a 3000 millisecond delay. The ZERO_FDL state is used to send zero targets to both FDLs, effectively parking them at mid range.

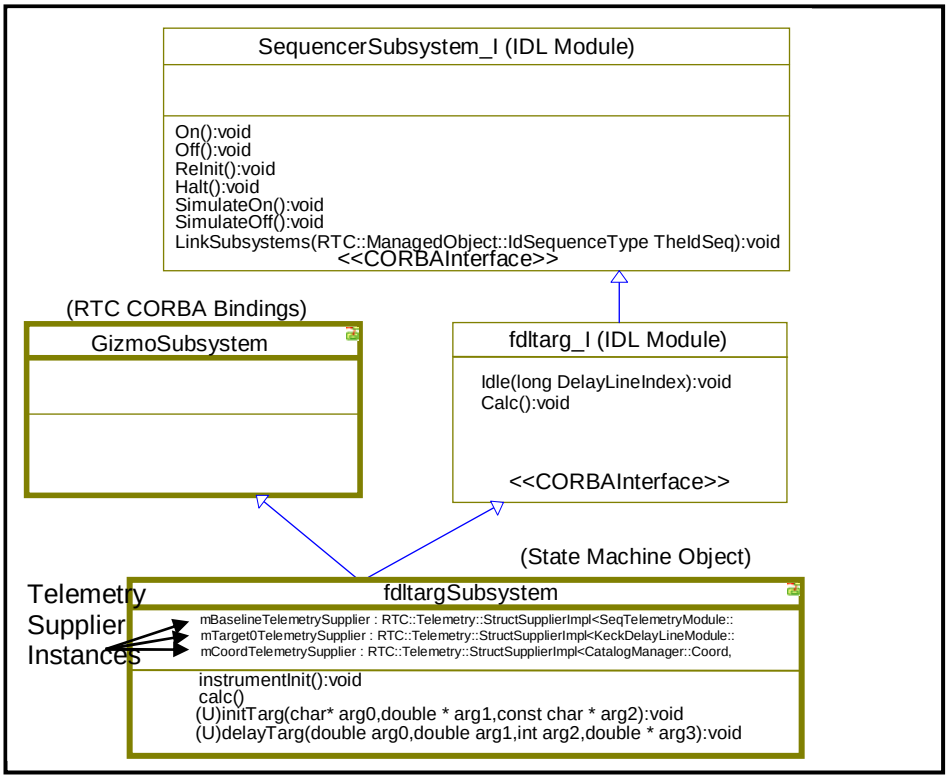

Fig. 7 Fast Delay Line Target Generator (fdltargSubsystem) static class diagram

5. LESSONS LEARNED

During the past several years that we have been exposed to CASE tool technology we *believe* a few things have been learned along the way. What follows is a list of lessons learned from our efforts to date, but there are without question more lessons to be learned about this technology:

1. It is hard to integrate multiple infrastructures and even harder to integrate them into a CASE environment. Although Rhapsody has hooks for adding external infrastructure dependencies, this was by no means easy to figure out. Corollary: On new CASE tool development projects do not mix, and reverse engineering is hard.
2. "Old programming habits die hard"[**]. It is hard to learn the graphical/coding UML methodology of the CASE tool environment if you have been a line-by-line coder for years. Corollary: The tool is powerful and useful but the learning curve is steep.
3. Maintainability of code is *vastly* improved. Because C++ code is coupled to a UML graphical model it is more understandable. The use of UML in this manner automatically limits the amount of reverse engineering of code required later in the software life cycle.
4. A CASE tool's real strength lies in its ability to standardize software development across a large team of engineers. However, using Rhapsody as an individual does have benefits. The state machine-based design can be rapidly changed graphically with only a minimum amount of coding required. Corollary: Standardized auto-generated code leads to better infrastructure.
5. When planning a project, consider a process for development and then a tool. Corollary: For CASE tools to be significantly beneficial requires an overall commitment and investment.
6. Version control of UML models between only a few developers can be is problematic. Corollary: When picking a CASE tool there should be good model collaboration and version control infrastructure within the tool.

[**] Douglas Schmidt, Using Design Patterns, Frameworks & CORBA, January 23-25, 2002, UCLA Extension Course.

7. One of the original attractive features of Rhapsody was animation of state machine functionality but this was never completely functional on the Solaris operating system, so the more traditional gdb was used for debugging. Corollary: Features in a tool may look good during a sales demo, but be careful!
8. When starting, one should evaluate and not guess at a CASE tool to use; there are lots of choices today.
9. Templates were not included with the Rhapsody case tool so we implemented them externally and hooked them into the tool. This is a kludge and is confusing to a new developer. This effectively defeats rule 3 above.
10. The urge to use many of Rhapsody's framework features has sometimes overwhelmed the wiser inclination to use more standardized tools such as the STL (C++ Standard Template Library).

6. CONCLUSION

We have implemented a multi-threaded state machine based, high-level, control capability for the Keck Interferometer. The main difference between interferometer sequencers comes down to multiplicities of baselines and instrument subsystems. The IF Sequencer subsystem (state-machine) classes have been designed to be highly configurable and flexible while reusing a basic state-machine initialization and error handling scheme.

The CASE tool approach is superior to hand line-by-line coding since it promotes both a graphic design technique and process driven approach to software design. Furthermore, an improvement in design and maintenance over the life-cycle is realized. Although dealing with third party infrastructures outside of the Rhapsody tool is problematic at first, after a significant effort at integration it becomes easy to graphically add states and additional software architecture. Rhapsody is an excellent tool, especially for state machine design and the problem fits nicely into the infrastructures provided by I-logix and the JPL developed RTC toolkit.

ACKNOWLEDGEMENT

The work performed here was conducted at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. The authors would like to thank the following people who contributed to this effort: Mark Colavita for helpful definition and support during the evaluation of the CASE tool; Thomas Lockhart and Kevin Tsubota for assistance resolving various software issues.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

REFERENCES

1. Andrew Booth, et. El., Overview of the control system for the Keck Interferometer, SPIE Advanced Telesope and Instrumentation Control Software II Conference 4848, Waikoloa, HI. August 2002.
2. T. Lockhart, RTC: a distributed real–time control system toolkit, SPIE Advanced Telesope and Instrumentation Control Software II Conference 4848, Waikoloa, HI. August 2002.
3. Erich Gamma, et. El., Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.
4. I-logix website http://www.ilogix.com/ has Rhapsody product information and white papers.
5. Philip C. Irwin, R. L. Johnson, Real-time control using an open source RTOS, SPIE Advanced Telescope and Instrumentation Control Software II Conference 4848, Waikoloa, HI, August 2002.
6. Michi Henning and Steve Vinoski, Advanced CORBA Programming with C++, Addison Wesley, 1999.
7. Interferometric Observation Planning Tool Suite, Michelson Science Center. http://msc.caltech.edu/software/.
8. Grady Booch, et. El. The Unified Modeling Language User Guide, Addison Wesley, 1999.